2021年度「プログラミング言語」配布資料 (4)

五十嵐 淳

2022年10月02日

1　2分探索木 in OCaml

1.1　永続的な2分探索木 in OCaml (ocaml/bst/)

1.1.1　ヴァリアントを使った2分木の表現

2分木の構造は menu のような再帰ヴァリアントを使うと簡単に定義することができる．

```
(* Defining the shape of trees *)
type tree =
    Lf       (* Leaf *)
  | Br of {  (* Branch *)
      left:  tree;
      value: int;
      right: tree;
    }
```

コメント[1]にあるように Lf が葉を表すコンストラクタ，Br が分岐を表すコンストラクタとなっており，Br には2つの部分木と整数のレコードが付加される．具体的な2分木は例えば

```
let t1 = Br {left = Lf; value = 10; right = Lf}
let t2 = Br {left = Lf; value = 25; right = Lf}
let t3 = Br {left = t1; value = 15; right = t2}
let t4 = Br {left = Lf; value = 60; right = Lf}
let t5 = Br {left = Lf; value = 48; right = t4}
let t6 = Br {left = t3; value = 30; right = t5}
```

のようにして作ることができる．(2分探索木を構成する Java プログラムと比べてみよう．)

2分木に関する関数は基本的に，木を引数として，その構造で場合分けを行い，Br の場合では，部分木について再帰呼び出しを行うような形の関数定義になる．

```
let rec f(t, ...) =
  match t with
    Lf -> ...
```

[1] OCaml ではコメントを (* と *) の間に書く．Java の /* ... */ と同様，コメント中に改行があってもよい．Java の // のような一行コメントのための記法は用意されていない．

```
  | Br {left=l; value=v; right=r} ->
      ... f(l, ...) ... f(r, ...)
```

変数 l が left フィールドの値，v が value フィールド，r が right フィールドの値につける名前となっている．パターンマッチのおかげで，部分木や整数の取り出しが簡潔に記述できている．

1.1.2 探索

探索のための関数 find は 00 2 分探索木 の擬似コード，もしくは，Java の static メソッドを使った実装によく似た形になる．

```
(* (Recursive) function find, which returns whether given integer n exists in BST t *)
let rec find(t, n) =
  match t with
    Lf -> false
  | Br {left=l; value=v; right=r} ->
     if n = v then true
     else if n < v then find(l, n)
     else (* n > v *)   find(r, n)
```

1.1.3 挿入

挿入も既にみた擬似コードとほぼ同じである．葉の場合には新しい節点を Br {left=Lf; value=n; right=Lf} を使って作っている．再帰呼び出しを行う際にも，挿入して得られた部分木を Br {...} に入れて，新しい木を返している．

```
(* (Recursive) function insert, which, given BST t and a new element n, returns
   a new binary search tree with n *)
let rec insert(t, n) =
  match t with
    Lf -> Br {left=Lf; value=n; right=Lf}
  | Br {left=l; value=v; right=r} ->
     if n = v then t
     else if n < v then Br {left=insert(l, n); value=v; right=r}
     else (* n > v *)   Br {left=l;            value=v; right=insert(r, n)}
```

1.1.4 削除

削除も，ここまでが理解できていれば簡単である．まずは，2 分探索木中の最小値を求める関数 min を定義する．

```
(* Function min, which, given BST t, returns the minimum value stored in t.
   If t is empty, it returns min_int. *)
let rec min t =
  match t with
    Lf -> min_int  (* The smallest representable integer, which is predefined *)
```

```
  | Br {left=Lf; value=v; right=_} -> v
  | Br {left=l ; value=_; right=_} -> min l
```

6 行目のパターンは，t が Br であり，**かつ，左の部分木が Lf であるような場合**を表している．また，ワイルドカードパターンを使って，右の部分木には興味がないことを示している．7 行目のパターンは，6 行目でマッチしなかった場合にのみ使われるので，必然的に l は Lf ではない，すなわち Br であることになる．

実は，レコードのパターンで right=_ のようにフィールドをまるごと捨てる場合は，そのフィールドを全く書かなくてもよく，

```
  | Br {left=Lf; value=v} -> v
  | Br {left=l} -> min l
```

または，

```
  | Br {left=Lf; value=v} -> v
  | Br {left=l; _} -> min l
```

と書いてもよい．後者のワイルドカードは，値を捨てるというよりも，残りのフィールド全体を捨てるという意味で使われている．“; _” があってもなくてもよいのだが，left 以外のフィールドがあることをプログラマが意識している，という点で好ましいスタイルのようだ．

さて，最後に，delete である．

```
(* (Recursive) function delete, which, given BST t and an element n to
   be deleted, returns a new binary search tree without n.  If n is not
   stored in t, it returns t as it is. *)
let rec delete(t, n) =
  match t with
    Lf -> t
  | Br {left=l; value=v; right=r} ->
      if n = v then
        match l, r with
          Lf,   Lf   -> Lf
        | Br _, Lf   -> l
        | Lf,   Br _ -> r
        | Br _, Br _ ->
            let m = min r in
            Br {left=l; value=m; right=delete(r, m)}
      else if n < v then Br {left=delete(l, n); value=v; right=r}
      else (* n > v *)   Br {left=l;            value=v; right=delete(r, n)}
```

9 行目からの match 式のように，ふたつ以上の値に対して同時に場合分けをしたい場合には，match ... with に式をコンマで並べることができる．パターンの方もそれに応じて式の数だけコンマで区切ったパターンを並べることになる．この機能のおかげで，左右の部分木が葉か節点かどうかの 4 通りの場合分けが，match を入れ子にすることなくすっきり書けている．

上で，レコードのパターンでフィールドを捨てる場合は，そのフィールドを全く書かなくてもよいことを説明

したが，一方で，パターン `Br _` の `_` は，今回のように例え全フィールドが不要な場合でも，省略できない．これは `Br` が引数を必要とすることを見落としているのはまずいという判断からであろう．(ここらへん，少し設計が一貫していない気もするが，レコードのフィールドを省略できることが例外的な感じがする．)

1.2 短命な 2 分探索木 in OCaml (ocaml/bstMutable/)

ここで一旦前の資料の変更可能データ構造に関する節を読むべし．

1.2.1 ヴァリアントを使った 2 分木の表現

短命な 2 分木は，`mutable` を使ってレコードの各フィールドを変更可能だと宣言するだけで表現することができる[2]．

```
(* Defining the shape of trees *)
type tree =
    Lf       (* Leaf *)
  | Br of {  (* Branch *)
      mutable left:  tree;  (* all fields are mutable *)
      mutable value: int;
      mutable right: tree;
    }
```

具体的な 2 分木の作り方は永続的な場合と全く同じである．

```
let t1 = Br {left = Lf; value = 10; right = Lf}
let t2 = Br {left = Lf; value = 25; right = Lf}
let t3 = Br {left = t1; value = 15; right = t2}
let t4 = Br {left = Lf; value = 60; right = Lf}
let t5 = Br {left = Lf; value = 48; right = t4}
let t6 = Br {left = t3; value = 30; right = t5}
```

1.2.2 探索

探索はもともと 2 分木データを読むだけで構造を変更するわけではないので定義は immutable な場合と全く変わらない．

```
let rec find(t, n) =
  match t with
```

[2]Java の場合，変更可能なインスタンス変数については何も書かなくてよく，変更を禁止する時に `final` と書いたのに対し，OCaml は変更可能にするために `mutable` という追記が必要なところに，どういうプログラミングを推奨しているかが垣間見える．

```
    Lf -> false
  | Br {left=l; value=v; right=r} ->
     if n = v then true
     else if n < v then find(l, n)
     else (* n > v *)   find(r, n)
```

1.2.3 挿入

挿入については，再帰呼び出しをした結果得られた新しい 2 分木を，フィールド更新を使って `left` や `right` フィールドに書き込んでいる部分が永続版との違いとなる (この違いは Java の場合と本質的に同じである).

```
(* (Recursive) function insert, which, given BST t and a new element n, returns
   a new binary search tree with n *)
let rec insert(t, n) =
  match t with
    Lf -> Br {left=Lf; value=n; right=Lf}
  | Br br ->
     if n = br.value then t
     else if n < br.value then (br.left <- insert(br.left, n); t)
     else (* n > br.value *)   (br.right <- insert(br.right, n); t)
```

木の種類についての場合わけで `Br` の場合に，部分木や格納された整数をパターンマッチで取り出さず，それらをまとめたレコードに `br` という名前をつけているが，これはフィールド変更式はレコード全体を指定する必要があるためである．上の定義では，各フィールドをドット記法で取り出しているが，実はレコード全体とフィールドの値にパターンを使って，同時に名前をつけることもできる．その場合は，

```
  match t with
   ...
  | Br ({left=l; value=v; right=r} as br) ->
     if n = v then t
     else if n < v then (br.left <- insert(l, n); t)
     else (* n > v *)   (br.right <- insert(r, n); t)
```

と書くことができる．`(... as br)` の部分は **as パターン**と呼ばれる記法で，マッチ対象の値 (ここではコンストラクタ `Br` に付加されたレコード) に対して `...` の部分に書かれたパターンマッチをしつつ，全体には `br` という名前をつけて，`->` の後を計算することができる．

最後の `if` の `then` 部，`else` 部に括弧がついているのは，セミコロンよりも `if`〜`then`〜`else` の結合が強いためである．(より詳しくは OCaml 爆速入門 (その 3, 制御構造) を参照のこと．)

つまり，例えば `(br.right <- insert(r, n); t)` の外側の括弧がないと，

```
     else (if n < v then (br.left <- insert(br.left, n); t)
                    else br.right <- insert(br.right, n));
     t
```

のように最後の t が if の一部とみなされなくなってしまう．その結果，then 部は t つまり tree 型の式が書かれているのに，else はフィールド変更式，すなわち unit 型の式になって分岐で型が違ってしまうため，

```
Error: This expression has type unit but an expression was expected of type tree
```

のようなエラーが出てくる．(“This expression”は br.right <- ... を示している．)

一方，then 部の括弧を外すと，セミコロン直後で if が一旦切れるように解釈されてしまうため，else が出てくるあたりで構文エラーになってしまう．

1.2.4 削除

さて，最後は削除関数 delete である．まず補助関数の min は全く変更する必要がないので省略する．

```
(* (Recursive) function delete, which, given BST t and an element n to
   be deleted, returns a new binary search tree without n.  If n is not
   stored in t, it returns t as it is. *)
let rec delete(t, n) =
  match t with
    Lf -> t
  | Br br ->
     if n = br.value then
       match br.left, br.right with
         Lf,   Lf   -> Lf
       | Br _, Lf   -> br.left
       | Lf,   Br _ -> br.right
       | Br _, Br _ ->
          let m = min br.right in
          br.value <- m;
          br.right <- delete(br.right, m);
          t
     else if n < br.value then (br.left <- delete(br.left, n); t)
     else (* n > br.value *)   (br.right <- delete(br.right, n); t)
```

これも Java 版と比べてみるとよいだろう．

構文について注意点をあげておく．let や match よりもセミコロンは結合が強いため，| Br _, Br _ -> ... や let m = ... in ... の範囲は 17 行目の t までとなる (15，16 行目のセミコロンで切れることはない)．そのため，14〜17 行目を括弧や begin〜end で括る必要はない．let や match はできるだけ右に延ばして読む，と覚えておくとよい．一方，match が入れ子になるときは注意が必要で，

```
match ... with
  A -> ...
| B -> match ... with
         C -> ...
       | D -> ...
| E -> ...
```

と書いても，| E -> ... の部分は内側の match の一部だと解釈されてしまう (match はできるだけ後ろに延ばして読むため)．よって，内側の match を C と D の場合だけで終わらせたい場合には，

```
match ... with
  A -> ...
| B -> (match ... with
          C -> ...
        | D -> ...)
| E -> ...
```

と括弧 (または begin と end) で括る必要がある．